16

مدخل إلى لغة التظليل عالية المستوى

سنصف في هذا الفصل لغة التظليل عالية المستوى (High-Level Shading Language) التي سنستخدمها في برمجة مظللات الرؤوس والعنصورات ضمن الفصول الثلاثة التالية. باختصار نقول أن مظللات الرؤوس والعنصورات عبارة عن برامج صغيرة خاصة نكتبها نحن، وتُنفَّذ ضمن وحدة معالجة الرسوميات (GPU أو graphic processing unit) الموجود في بطاقة الرسوميات، حيث تستعيض هذه البرامج جزءاً من خط المعالجة ذو الوظيفة الثابتة. نحصل من خلال استبدال قسم من خط المعالجة ذو الوظيفة الثابتة ببرنامجنا المخصص (المظلل) على كمّ هائل من المرونة في إنجاز التأثيرات الرسومية، وبالتالي لم نعد مقيدين بالعمليات الثابتة مسبقة التعريف.

نحتاج من أجل كتابة برامج التظليل إلى لغة لكتابتها. تمت كتابة المظللات في DirectX 8.x بلغة تظليل تجميعية (assembly) منخفضة المستوى، ولكن لحسن الحظ لسنا مجبرين لكتابة المظللات بلغة التجميع بعد الآن، لأن DirectX 9 قد وفّر لغة التظليل عالية المستوى التي يمكن أن نستخدمها لكتابة المظللات. إن لاستخدام HLSL بدلاً من لغة التجميع لكتابة برامج التظليل نفس فوائد استخدام لغة عالية المستوى (مثل ++C أو Pascal) بدلاً من لغة التجميع لكتابة التطبيقات. هذه الفوائد هي:

- إنتاجية أكبر: كتابة البرامج بلغة عالية المستوى أسرع وأسهل من كتابتها بلغة

منخفضة المستوى. فهنا نقضي وقتاً أكبر في التركيز على الخوارزميات لا في كتابة الشيفرة.

- قابلية أحسن للقراءة: البرامج المكتوبة بلغة عالية المستوى أسهل للقراءة، الأمر الذي يؤدي إلى أن تنقيح وصيانة البرامج المكتوبة بلغة عالية المستوى أسهل.
- تولد المترجمات (في أغلب الأحيان) شيفرة تجميع أكثر فعالية من شيفرة التجميع المكتوبة يدوياً.
- نستطيع من خلال استعمال مترجم HLSL أن نترجم شيفرتنا إلى أي إصدار متاح من المظللات، أما عند استخدام لغة التجميع فيجب الالتزام بكتابة شيفرة لكل إصدار نريده.

إن لغة HLSL تشبه كثيراً صيغة لغة C أو C++، لذا ليس من الصعب الانتقال إلى الصياغة بهذه اللغة.

أخيراً نذكر بأنه يجب التبديل إلى الجهاز REF من أجل أمثلة المظللات إذا لم تكن بطاقة الرسوميات تدعم مظللات الرؤوس والعنصورات. إن استخدام الجهاز REF يعني أن أمثلة المظللات ستُنفَّذ ببطء ولكن مع إظهار النتائج الصحيحة بحيث يمكننا التأكد من صحة الشيفرة التي نكتبها.

ملاحظة: يتم تنفيذ مظللات الرؤوس برمجياً من خلال معالجة الرؤوس برمجياً (D3DCREATE_SOFTWARE_VERTEX_PROCESSING).

الأهداف

- تعلم كيفية كتابة وترجمة برنامج تظليل مكتوب بلغة HLSL.
- تعلم كيفية اتصال المعطيات في التطبيق مع برنامج المظلل.
- التآلف مع الصيغة والأنواع والتوابع الضمنية للغة HLSL.

16.1: كتابة مظلل بلغة HLSL

يمكننا أن نكتب شيفرة مظللات HLSL مباشرة ضمن الملفات المصدرية لتطبيقنا على شكل سلسلة رمزية طويلة. ولكن من الملائم فصل شيفرة المظلل عن شيفرة التطبيق،

لذلك سنكتب المظللات ضمن برنامج مثل المفكرة (Notepad) ونحفظها كملفات نصية عادية (برموز ASCII)، ثم نستخدم التابع D3DXCompileShaderFromFile (المشروح في الفقرة 16.2.2) من أجل ترجمة المظللات.

كمقدمة نورد مظلل الرؤوس البسيط التالي والمكتوب بلغة HLSL والذي تم حفظه في ملف نصي اسمه Transform.txt (ملفات المشروع الكامل موجودة في القرص المرفق تحت عنوان Transform). يقوم هذا المظلل بتحويل الرؤوس نقطياً باستخدام مصفوفة مركّبة من العرض والإسقاط كما يضع مركبة اللون المنتثر للرأس باللون الأزرق.

ملاحظة: يستخدم هذا المثال مظلل رؤوس من أجل التوضيح، وليس هناك داعٍ للقلق حول ما يُفترض أن يقوم به هذا المظلل الآن لأن ذلك سيُشرح في الفصل القادم. أما الغاية الحالية فهي التآلف مع صيغة وتنسيق برنامج HLSL.

```
// File: transform.txt
// Desc: Vertex shader that transforms a vertex by the view and
// projection transformation, and sets the vertex color to blue.

//
// Globals
//

// Global variable to store a combined view and projection
// transformation matrix.  We initialize this variable
// from the application.
matrix ViewProjMatrix;

// Initialize a global blue color vector.
vector Blue = {0.0f, 0.0f, 1.0f, 1.0f};

//
// Structures
//

// Input structure describes the vertex that is input
// into the shader.  Here the input vertex contains
// a position component only.
struct VS_INPUT
{
    vector position  : POSITION;
```

```
};

// Output structure describes the vertex that is
// output from the shader.  Here the output
// vertex contains a position and color component.
struct VS_OUTPUT
{
    vector position : POSITION;
    vector diffuse  : COLOR;
};

//
// Main Entry Point, observe the main function
// receives a copy of the input vertex through
// its parameter and returns a copy of the output
// vertex it computes.
//

VS_OUTPUT Main(VS_INPUT input)
{
    // zero out members of output
    VS_OUTPUT output = (VS_OUTPUT)0;

    // transform to view space and project
    output.position  = mul(input.position, ViewProjMatrix);

    // set vertex diffuse color to blue
    output.diffuse = Blue;

    return output;
}
```

16.1.1: المتحولات العامة

أولاً نقوم بتعريف متحولين عامين:

```
matrix ViewProjMatrix;
vector Blue = {0.0f, 0.0f, 1.0f, 1.0f};
```

المتحول الأول ViewProjMatrix من النوع matrix (والذي هو نوع لمصفوفة 4×4 معرف ضمنياً في HLSL)، سيخزن هذا المتحول مصفوفة العرض والإسقاط المركبة بحيث يصف هذين التحويلين النقطيين معاً. بهذه الطريقة سيكون علينا إجراء عملية ضرب شعاع بمصفوفة واحدة بدلاً من عمليتين. لاحظ بأننا لا نهيء هذا المتحول في أي مكان

ضمن الشيفرة المصدرية للمظلل، لأننا نحدد قيمته من خلال الشيفرة المصدرية للتطبيق وليس في المظلل. إن عملية اتصال التطبيق مع برنامج المظلل مطلوبة كثيراً وسيتم شرحها في الفقرة 16.2.1.

المتحول الثاني Blue من النوع الضمني vector (والذي هو شعاع رباعي الأبعاد). نقوم بتهيئة مركبات هذا الشعاع لتمثل اللون الأزرق وذلك بمعاملته كشعاع لوني RGBA.

16.1.2: بنيتا الدخل والخرج

بعد التصريح عن المتحولات العامة نعرّف بنيتان خاصتان سندعوهما بنيتا الدخل والخرج. بالنسبة إلى مظللات الرؤوس تعرّف هاتان البنيتان معطيات الرأس التي يستخدمها المظلل كدخل وخرج.

```
struct VS_INPUT
{
    vector position  : POSITION;
};

struct VS_OUTPUT
{
    vector position : POSITION;
    vector diffuse  : COLOR;
};
```

ملاحظة: تعرف بنيتا الدخل والخرج لمظلل العنصورات معطيات عنصورة.

في هذا المثال يحتوي الرأس الذي يدخل إلى مظلل الرؤوس على مركبة للموضع فقط، أما الرأس الذي يعطيه مظلل الرؤوس كخرج فيحتوي على مركبة للموضع ومركبة للون.

تشير علامة التفصيل (:) إلى المعنى (semantic) الذي يستخدم من أجل تحديد استعمال المتحول. إن المعنى يشبه تنسيق الرؤوس المرن (FVF) لبنية رأس، مثلاً لدينا في VS_INPUT الحقل التالي:

```
vector position  : POSITION;
```

الصيغة ":POSITION" تنص على أن الشعاع position يُستعمل لوصف مكان رأس الدخل. لدينا مثال آخر حول ذلك، في البنية VS_OUTPUT يوجد:

```
vector diffuse  : COLOR;
```

حيث هنا ":COLOR" ينص على أن الشعاع يُستعمل من أجل وصف لون رأس الخرج. سنتحدث أكثر حول معرّفات الاستعمال المتاحة لاحقاً في الفصلين التاليين عند دراسة مظللات الرؤوس والعنصورات.

ملاحظة: من منظور منخفض المستوى، تقوم صيغة المعنى بربط أي متحول في المظلل بمسجل (register) في العتاد. أي أن متحولات الدخل تُربط مع مسجلات الدخل وتربط متحولات الخرج مع مسجلات الخرج. على سبيل المثال، يتصل الحقل في البنية VS_INPUT بمسجل مكان رأس الدخل، وبشكل مشابه يتصل diffuse مع مسجل لون معين لرأس الخرج.

16.1.3: تابع نقطة الدخول

لكل برنامج HLSL (كما في برنامج بلغة C++) نقطة دخول. في مثالنا حول مظلل الرؤوس سمينا تابع نقطة الدخول Main ولكننا لسنا ملزمين بهذه التسمية، إذ يمكن أن يأخذ تابع نقطة الدخول في مظلل أي اسم مقبول كاسم تابع. يجب أن يكون لتابع نقطة الدخول وسيطاً هو بنية الدخل والذي يُستخدم لتمرير رأس الدخل إلى المظلل، كما يجب على تابع نقطة الدخول أن يعيد بنية خرج والتي فيها يوجد الرأس المعالَج من قبل المظلل.

```
VS_OUTPUT Main(VS_INPUT input)
{
```

ملاحظة: في الواقع، ليس من الواجب استخدام بنيتا دخل وخرج. على سبيل المثال، قد ترى أحياناً صيغة مشابهة لما يلي خصوصاً في مظللات العنصورات:

```
float4 Main(in float2 base : TEXCOORD0,
            in float2 spot : TEXCOORD1,
            in float2 text : TEXCOORD2) : COLOR;
{
    ...
}
```

حيث تكون الوسطاء دخلاً للمظلل (في مثالنا المذكور نقوم بإدخال ثلاث إحداثيات نسيجية). يعيد المظلل لوناً وحيداً كخرج حيث تمت الإشارة إلى ذلك من خلال الصيغة ":COLOR" التي تلي رأس التابع. إن هذا التعريف يكافئ ما يلي:

```
struct INPUT
{
    float2 base : TEXCOORD0,
```

```
    float2 spot : TEXCOORD1,
    float2 text : TEXCOORD2)
};

struct OUTPUT
{
    float4 c : COLOR;
};

OUTPUT Main(INPUT input)
{
    ...
}
```

إن جسم تابع نقطة الدخول مسؤول عن حساب رأس الخرج اعتباراً من رأس الدخل. يقوم المظلل في هذا المثال بتحويل رأس الدخل إلى فضاء العرض وفضاء الإسقاط، ويجعل لون الرأس أزرق، ويعيد الرأس الناتج. أولاً نقوم بإنشاء نسخة من VS_OUTPUT ونضع في كل حقولها القيمة 0:

```
VS_OUTPUT output = (VS_OUTPUT)0;
```

ثم يجري المظلل تحويلاً نقطياً لمكان رأس الدخل من خلال ضربه بالمتحول ViewProjMatrix باستخدام التابع mul، الذي هو تابع ضمني يمكنه ضرب شعاع بمصفوفة وضرب مصفوفة بمصفوفة. نحفظ الشعاع الناتج عن التحويل النقطي في حقل الموضع ضمن بنية الخرج:

```
output.position = mul(input.position, ViewProjMatrix);
```

بعد ذلك نضع في حقل اللون المنتثر ضمن output القيمة BLUE:

```
output.diffuse = BLUE;
```

وأخيراً نعيد الرأس الناتج:

```
return output;
}
```

16.2: ترجمة مظلل مكتوب بلغة HLSL

16.2.1: جدول الثوابت

يوجد لكل مظلل جدول ثوابت يستخدم لتخزين متحولاته. توفر المكتبة D3DX لتطبيقنا إمكانية الولوج إلى جدول ثوابت المظلل من خلال الواجهة ID3DXConstantTable. يمكننا بواسطة هذه الواجهة تحديد قيم المتحولات الموجودة في الشيفرة المصدرية

للمظلل من خلال شيفرة تطبيقنا.

سنورد الآن لائحة مختصرة بالمناهج التي تحققها الواجهة ID3DXConstantTable. من أجل الحصول على اللائحة الكاملة بهذه المناهج راجع وثائق Direct3D.

16.2.1.1: الحصول على مقبض لثابت

من أجل تحديد قيمة متحول معين في المظلل من خلال شيفرة تطبيقنا نحتاج إلى طريقة للرجوع إليها. يتم الرجوع إلى متحول في المظلل من قبل تطبيقنا باستخدام مقبض من النوع D3DXHANDLE. يعيد المنهج التالي مقبضاً لمتحول في المظلل اعتماداً على اسمه:

```
D3DXHANDLE ID3DXConstantTable::GetConstantByName(
    D3DXHANDLE hConstant,    // scope of constant
    LPCSTR pName             // name of constant
);
```

حيث:

- **hConstant**: المقبض الذي يعرف البنية الأب التي تحوي المتحول الذي نود الحصول على مقبض له. مثلاً، إذا أردنا الحصول على مقبض لحقل معطيات وحيد في بنية معينة فعلينا أن نمرر مقبض البنية في هذا الوسيط. إذا كنا نريد الحصول على مقبض لمتحول موجود في أعلى مستوى عندها نمرر قيمة 0.
- **pName**: اسم المتحول المستخدم في الشيفرة المصدرية للمظلل والذي نود الحصول على مقبض له.

على سبيل المثال، لو كان اسم المتحول في المظلل هو ViewProjMatrix وكان موجوداً في أعلى مستوى عندها نكتب:

```
D3DXHANDLE h0;
h0 = ConstTable->GetConstantByName(0, "ViewProjMatrix");
```

16.2.1.2: تحديد قيم الثوابت

حالما يحصل تطبيقنا على مقبض يشير إلى متحول معين في شيفرة المظلل عندها نستطيع تحديد قيمة المتحول داخل تطبيقنا من خلال استخدام المناهج ID3DXConstantTable::SetXXX. حيث XXX، تستبدل باسم نوع يدل على نوع المتحول المراد تحديد قيمته. على سبيل المثال، إذا كان المتحول الذي نود تحديد قيمته هو نسق

من الأشعة فعندئذ يكون اسم المنهج SetVectorArray.

الصيغة العامة لمنهج ID3DXConstantTable::SetXXX هي على الشكل التالي:

```
HRESULT ID3DXConstantTable::SetXXX(
    LPDIRECT3DDEVICE9 pDevice,
    D3DXHANDLE hConstant,
    XXX value
);
```

- **pDevice**: مؤشر إلى الجهاز المخصص لجدول الثوابت.
- **hConstant**: مقبض يشير إلى المتحول الذي نريد تحديد قيمته.
- **value**: القيمة التي نريد وضعها في المتحول حيث أن XXX يستبدل باسم نوع المتحول الذي نريد تحديد قيمته. بالنسبة إلى بعض الأنواع (مثل: bool أو int أو float) نمرر نسخة من القيمة، وبالنسبة لبعض الأنواع الأخرى (مثل: vector أو matrix أو البنى) نمرر مؤشراً إلى القيمة.

عند تحديد قيم مصفوفات يأخذ المنهج SetXXX وسيطاً رابعاً إضافياً يحدد عدد عناصر المصفوفة. مثلاً، يكون رأس المنهج الذي يقوم بتحديد قيمة نسق من أشعة رباعية الأبعاد بالشكل التالي:

```
HRESULT ID3DXConstantTable::SetVectorArray(
    LPDIRECT3DDEVICE9 pDevice,
    D3DXHANDLE hConstant,
    CONST D3DXVECTOR4* pVector,
    UINT Count
);
```

تصف اللائحة التالية الأنواع التي يمكننا استخدامها من خلال الواجهة ID3DXConstantTable. بفرض أنه لدينا جهازاً اسمه Device ومقبض المتحول الذي نريد تحديد قيمته هو handle:

- **SetBool**: يستخدم لوضع قيمة بوليانية:

```
bool b = true;
ConstTable->SetBool(Device, handle, b);
```

- **SetBoolArray**: يستخدم لوضع نسق من القيم بوليانية:

```
bool b[3] = {true, false, true};
ConstTable->SetBoolArray(Device, handle, b, 3);
```

- **SetFloat**: يستخدم لوضع قيمة حقيقية:

```
float f = 3.14f;
ConstTable->SetFloat(Device, handle, f);
```

- **SetFloatArray**: يستخدم لوضع نسق من القيم الحقيقية:

```
float f[2] = {1.0f, 2.0f};
ConstTable->SetFloatArray(Device, handle, f, 2);
```

- **SetInt**: يستخدم لوضع قيمة صحيحة:

```
int x = 4;
ConstTable->SetInt(Device, handle, x);
```

- **SetIntArray**: يستخدم لوضع نسق من القيم الصحيحة:

```
int x[4] = {1, 2, 3, 4};
ConstTable->SetIntArray(Device, handle, x, 4);
```

- **SetMatrix**: يستخدم لوضع قيم مصفوفة 4×4:

```
D3DXMATRIX M(...);
ConstTable->SetMatrix(Device, handle, &M);
```

- **SetMatrixArray**: يستخدم لوضع قيم نسق من المصفوفات 4×4:

```
D3DXMATRIX M[4];

// ...initialize matrices.

ConstTable->SetMatrixArray(Device, handle, M, 4);
```

- **SetMatrixPointerArray**: يستخدم لوضع قيم نسق من مؤشرات تشير كل منها إلى مصفوفة 4×4:

```
D3DXMATRIX* M[4];

// ...allocate and initialize matrix pointers.

ConstTable->SetMatrixPointerArray(Device, handle, M, 4);
```

- **SetMatrixTranspose**: يستخدم لوضع قيمة منقول مصفوفة 4×4.

```
D3DXMATRIX M(...);
D3DXMatrixTranspose(&M, &M);
ConstTable->SetMatrixTranspose(Device, handle, &M);
```

- **SetMatrixTransposeArray**: يستخدم لوضع قيم نسق من منقولات لمصفوفات 4×4:

```
D3DXMATRIX M[4];
```

```
// ...initialize matrices and transpose them.
ConstTable->SetMatrixTransposeArray(Device, handle, M, 4);
```

- **SetMatrixTransposePointerArray**: يستخدم لوضع قيم نسق من المؤشرات يشير كل منها إلى منقول مصفوفة 4×4:

```
D3DXMATRIX M[4];
// ...allocate and initialize matrices and transpose them.
ConstTable->SetMatrixTransposePointerArray(Device, handle, M, 4);
```

- **SetVector**: يستخدم لوضع قيم متحول من النوع D3DXVECTOR4:

```
D3DXVECTOR4 v(1.0f, 2.0f, 3.0f, 4.0f);
ConstTable->SetVector(Device, handle, &v);
```

- **SetVectorArray**: يستخدم لوضع قيمة متحول عبارة عن نسق من الأشعة:

```
D3DXVECTOR4 v[3];
// ...initialize vectors
ConstTable->SetVectorArray(Device, handle, v, 3);
```

- **SetValue**: يستخدم لوضع قيمة من نوع ذو حجم غير مبين مثل بنية. مثلاً لاستخدام SetValue لوضع قيمة من النوع D3DXMATRIX نكتب:

```
D3DXMATRIX M(...);
ConstTable->SetValue(Device, handle, (void*)&M, sizeof(M));
```

16.2.1.3: تحديد قيم الثوابت الافتراضية

يقوم المنهج التالي بوضع الثوابت بقيمها الافتراضية، والتي هي القيم التي تم استخدامها في تهيئة الثوابت عندما تم التصريح عنها. يجب استدعاء هذا المنهج مرة واحدة أثناء إعداد التطبيق:

```
HRESULT ID3DXConstantTable::SetDefaults(
    LPDIRECT3DDEVICE9 pDevice
);
```

- **pDevice**: مؤشر إلى الجهاز المخصص لجدول الثوابت.

16.2.2: ترجمة مظلل بلغة HLSL

يتم ترجمة مظلل بعد كتابته في ملف نصي باستخدام التابع التالي:

```
HRESULT D3DXCompileShaderFromFile(
    LPCTSTR pSrcFile,
    CONST D3DXMACRO* pDefines,
    LPD3DXINCLUDE pInclude,
    LPCTSTR pFunctionName,
    LPCTSTR pTarget,
    DWORD Flags,
    LPD3DXBUFFER* ppShader,
    LPD3DXBUFFER *ppErrorMsgs,
    LPD3DXSHADER_CONSTANTTABLE *ppConstantTable
);
```

- **pSrcFile**: اسم الملف النصي الحاوي على الشيفرة المصدرية للمظلل الذي تريد ترجمته.
- **pDefines**: هذا الوسيط خياري لذا سنمرر فيه القيمة null حيثما ورد في الكتاب.
- **pInclude**: مؤشر إلى واجهة ID3DXInclude. هذه الواجهة مصممة بحيث يتم تحقيقها في التطبيق حتى نقوم بتجاوز سلوك التضمين الافتراضي. على العموم إن السلوك الافتراضي مناسب لنا لذا يمكننا تجاهل هذا الوسيط بوضع قيمة null فيه.
- **pFunctionName**: سلسلة رمزية تحدد اسم تابع نقطة الدخول. مثلاً إذا كان اسم تابع نقطة الدخول في المظلل Main عندها نمرر السلسلة "Main" في هذا الوسيط.
- **pTarget**: سلسلة رمزية تحدد إصدار المظلل الذي نريد ترجمة شيفرة HLSL المصدرية به. إن إصدارات مظللات الرؤوس المتاحة: vs_1_1 وvs_2_0 وvs_2_sw. إصدارات مظللات العنصورات المتاحة: ps_1_1 وps_1_2 وps_1_3 وps_1_4 وps_2_0 وps_2_sw. مثلاً، إذا أردنا ترجمة مثال مظلل الرؤوس المعروض في أول الفصل إلى الإصدار 2.0 عندها نمرر vs_2_0 في هذا الوسيط.

ملاحظة: إن القدرة على الترجمة إلى إصدارات مختلفة للمظللات هي إحدى الفوائد الرئيسية من استخدام لغة HLSL بدلاً من لغة التجميع. يمكننا من خلال استخدام لغة HLSL تصدير أي مظلل تقريباً إلى أي إصدار ببساطة عن طريق إعادة ترجمته إلى الإصدار المطلوب، أما عند استخدام لغة التجميع فعلينا تصدير الشيفرة يدوياً.

- **Flags**: عبارة عن مجموعة من أعلام خيارية للترجمة (ضع قيمة 0 لتحديد عدم

وجود أعلام) الخيارات المتاحة هي:

- **D3DXSHADER_DEBUG:** يأمر المترجم بكتابة معلومات التنقيح.
- **D3DXSHADER_SKIPVALIDATION:** يأمر المترجم بعدم إجراء أي فحص للشيفرة (يجب أن لا يستخدم هذا العلم مع أي مظلل إلا إذا كنت واثقاً من أنه يعمل).
- **D3DXSHADER_SKIPOPTIMIZATION:** يأمر المترجم بعدم إجراء أي اختزال للشيفرة. عملياً يستخدم هذا فقط من أجل التنقيح حيث لا ترغب أن يقوم المترجم بتغيير الشيفرة بأي شكل كان.

- **ppShader**: يعيد مؤشراً إلى واجهة ID3DXBuffer تحتوي على شيفرة المظلل المترجمة. تستخدم شيفرة المظلل المترجمة هذه بعد ذلك كوسيط لتابع آخر ينشئ مظلل الرؤوس أو العنصورات الفعلي.
- **ppErrorMsgs**: يعيد مؤشراً إلى واجهة ID3DXBuffer تحتوي على سلسلة رمزية فيها شيفرات ورسائل الخطأ.
- **ppConstantTable**: يعيد مؤشراً إلى واجهة ID3DXConstantTable تحتوي على معطيات جدول ثوابت هذا المظلل.

فيما يلي مثال حول استدعاء D3DXCompileShaderFromFile:

```
//
// Compile shader.
//

ID3DXConstantTable* TransformConstantTable = 0;
ID3DXBuffer* shader      = 0;
ID3DXBuffer* errorBuffer = 0;

hr = D3DXCompileShaderFromFile(
    "transform.txt",
    0,
    0,
    "Main",  // entry point function name
    "vs_2_0",// shader version to compile to
    D3DXSHADER_DEBUG,
    &shader,
    &errorBuffer,
    &TransformConstantTable);

// output any error messages
if( errorBuffer )
```

```
{
    ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(), 0, 0);
    d3d::Release<ID3DXBuffer*>(errorBuffer);
}

if(FAILED(hr))
{
    ::MessageBox(0, "D3DXCreateEffectFromFile() - FAILED", 0, 0);
    return false;
}
```

16.3: أنواع المتحولات

ملاحظة

هنالك بعض الأنواع الغرضية الضمنية في HLSL بالإضافة إلى الأنواع الموصوفة في الفقرات التالية (مثل غرض texture) لكن بما أن هذه الأنواع الغرضية تستخدم بشكل رئيسي فقط في إطار عمل التأثيرات لذا نؤجل دراستهم إلى الفصل التاسع عشر.

16.3.1: الأنواع السلمية

تزودنا لغة HLSL بالأنواع السلمية التالية:

- **bool**: يأخذ قيمة true أو false (لاحظ أن true وfalse هما كلمتان محجوزتان ضمن لغة HLSL).
- **int**: عدد صحيح مؤشر بعرض 32 بت.
- **half**: عدد ذو فاصلة عائمة بعرض 16 بت.
- **float**: عدد ذو فاصلة عائمة بعرض 32 بت.
- **double**: عدد ذو فاصلة عائمة بعرض 64 بت.

ملاحظة

قد لا تدعم بعض الأنظمة الأنواع: int وhalf وdouble. في هذه الحالة يتم استخدام النوع float بدلاً عنها.

16.3.2: أنواع الأشعة

يوجد في لغة HLSL أنواع الأشعة الضمنية التالية:

- **vector**: شعاع رباعي الأبعاد تكون كل مركبة فيه من النوع float.
- **vector<T, n>**: شعاع ذو n بعد حيث تكون كل مركبة فيه من النوع السلمي T. البعد n يجب أن يكون بين 1 و4. هنا مثال عن شعاع ثنائي الأبعاد مركباته أعداد حقيقية ذات دقة مضاعفة:

```
vector<double, 2> vec2;
```

يمكننا الولوج إلى أي مركبة في شعاع باستخدام الصيغة الدليلية للأنساق. مثلاً لوضع قيمة في المركبة (i) من الشعاع vec نكتب:

```
vec[i] = 2.0f;
```

كما يمكننا أيضاً الولوج إلى مركبات شعاع vec مثلما يتم الولوج إلى حقول أي بنية عن طريق أسماء المركبات المعرفة التالية: a, b, g, r, w, z, y, x.

```
vec.x = vec.r = 1.0f;
vec.y = vec.g = 2.0f;
vec.z = vec.b = 3.0f;
vec.w = vec.a = 4.0f;
```

حيث تشير الأسماء a, b, g, r إلى نفس المركبات التي تشير إليها الأسماء w, z, y, x على الترتيب. حيث أنه من الأفضل استخدام صيغة RGBA لتمثيل الألوان لأن ذلك يؤكد أن الشعاع يمثل لوناً.

يمكننا بدلاً من ذلك استخدام الأنواع المسبقة التعريف الأخرى التي تمثل أشعة ثنائية الأبعاد وثلاثية الأبعاد ورباعية الأبعاد:

```
float2 vec2;
float3 vec3;
float4 vec4;
```

ليكن لدينا الشعاع $u = (u_x, u_y, u_z, u_w)$، ونود نسخ مركبات u إلى شعاع v ليصبح $v = (v_x, v_y, v_z, v_w)$. يكون الحل البديهي هو نسخ كل مركبة من u إلى v كل على حدة حسب الحاجة. ولكن تزودنا لغة HLSL بصيغة خاصة لإجراء هذا النوع الخاص من النسخ تدعى (swizzle):

```
vector u = {1.0f, 2.0f, 3.0f, 4.0f};
vector v = {0.0f, 0.0f, 5.0f, 6.0f};

v = u.xyyw; // v = {1.0f, 2.0f, 2.0f, 4.0f }
```

لا يتوجب علينا نسخ كل المركبات عند نسخ الأشعة. مثلاً يمكننا نسخ المركبتين y, x فقط كما يتضح من الشيفرة التالية:

```
vector u = {1.0f, 2.0f, 3.0f, 4.0f};
vector v = {0.0f, 0.0f, 5.0f, 6.0f};

v.xy = u; // v = {1.0f, 2.0f, 5.0f, 6.0f }
```

16.3.3: أنواع المصفوفات

يوجد في لغة HLSL أنواع المصفوفات الضمنية التالية:

- **matrix:** مصفوفة 4×4 حيث كل حجرة فيها من النوع float.
- **matrix<T, m, n>:** مصفوفة m×n حيث كل حجرة فيها من النوع السلمي T يجب أن تكون أبعاد المصفوفة n, m بين 1 و 4. هنا مثال عن مصفوفة 2×2 من الأعداد الصحيحة:

```
matrix<int, 2, 2> m2x2;
```

يمكننا بدلاً من ذلك تعريف مصفوفة m×n (حيث n, m يتروحان بين 1 و 4) باستخدام الصيغة التالية:

```
floatmxn matmxn;
```

أمثلة:

```
float2x2 mat2x2;
float3x3 mat3x3;
float4x4 mat4x4;
float2x4 mat2x4;
```

ملاحظة: ليس من اللازم أن يكون النوع هو float دائماً. حيث يمكننا استخدام أي نوع آخر. مثلاً، لاستخدام أعداد صحيحة نكتب:

```
int2x2 i2x2;
int3x3 i3x3;
int2x4 i2x4;
```

يمكننا الولوج إلى أي حجرة في مصفوفة باستخدام الصيغة الدليلية المزدوجة للأنساق. مثلاً، لوضع قيمة في الحجرة (ij) من المصفوفة M نكتب:

```
M[i][j] = value;
```

كما يمكننا الإشارة إلى حجرات المصفوفة M مثلما يتم الولوج إلى حقول أي بنية. فيما يلي نذكر أسماء الحجرات المعرفة:

- بدءاً من الواحد:

```
M._11 = M._12 = M._13 = M._14 = 0.0f;
```

```
M._21 = M._22 = M._23 = M._24 = 0.0f;
M._31 = M._32 = M._33 = M._34 = 0.0f;
M._41 = M._42 = M._43 = M._44 = 0.0f;
```

- بدءاً من الصفر:

```
M._m00 = M._m01 = M._m02 = M._m03 = 0.0f;
M._m10 = M._m11 = M._m12 = M._m13 = 0.0f;
M._m20 = M._m21 = M._m22 = M._m23 = 0.0f;
M._m30 = M._m31 = M._m32 = M._m33 = 0.0f;
```

قد نود الإشارة إلى شعاع سطر معين في مصفوفة. يمكننا ذلك من خلال استخدام الصيغة الدليلية المفردة للأنساق. مثلاً، للإشارة إلى شعاع السطر (i) من المصفوفة M نكتب:

```
vector ithRow = M[i];
```

ملاحظة

يمكننا تهيئة المتحولات بلغة HLSL عن طريق الصيغتين التاليتين:

```
vector u = {0.6f, 0.3f, 1.0f, 1.0f};
vector v = {1.0f, 5.0f, 0.2f, 1.0f};
```

أو بشكل مكافئ يمكننا استخدام صيغة الباني:

```
vector u = vector(0.6f, 0.3f, 1.0f, 1.0f);
vector v = vector(1.0f, 5.0f, 0.2f, 1.0f);
```

وفيما يلي أمثلة أخرى:

```
float2x2 f2x2 = float2x2(1.0f, 2.0f, 3.0f, 4.0f);
int2x2 m = {1, 2, 3, 4};
int n = int(5);
int a = {5};
float3 x = float3(0, 0, 0);
```

16.3.4: الأنساق

يتم التصريح عن نسق من نوع معين باستخدام صيغة لغة ++C المألوفة، مثلاً:

```
float M[4][4];
half p[4];
vector v[12];
```

16.3.5: البنى

يتم تعريف البنى تماماً كما تعرف في لغة C++. لكن لا يمكن للبنى في لغة HLSL أن تمتلك مناهج. فيما يلي مثال عن بنية بلغة HLSL:

```
struct MyStruct
{
    matrix T;
    vector n;
    float f;
    int x;
    bool b;
};
MyStruct s;
s.f = 5.0f;
```

16.3.6: الكلمة المحجوزة typedef

إن وظيفة الكلمة المحجوزة typedef في لغة HLSL هي نفسها تماماً كما في لغة C++. مثلاً يمكننا إطلاق اسم point على النوع vector<float, 3> باستخدام الصيغة التالية:

```
typedef vector<float, 3> point;
```

عندئذ بدلاً من أن نكتب:

```
vector<float, 3> myPoint;
```

يمكننا فقط أن نكتب:

```
point myPoint;
```

هنا مثالين آخرين يبينان استخدام الكلمة المحجوزة typedef لتعريف نوع ثابت ونسق:

```
typedef const float CFLOAT;
typedef float point2[2];
```

16.3.7: سوابق المتحولات

يمكن استخدام الكلمات المحجوزة التالية كسابقة عند التصريح عن متحول:

- **static**: إذا كان متحول ما مسبوقاً بالكلمة المحجوزة static فهذا يعني بأنه لن يكون مرئياً خارج المظلل. بتعبير آخر سيكون هذا المتحول محلياً في المظلل.

  أما إذا تم إسباق متحول محلي بالكلمة المحجوزة static عندها سيسلك نفس سلوك متحول محلي static في لغة C++. أي يتم تهيئته مرة واحدة عندما يتم تنفيذ التابع

ويبقى محافظاً على قيمته خلال كافة استدعاءات هذا التابع. إذا لم تتم تهيئة المتحول عندها تتم تهيئته آلياً بالقيمة 0.

```
static int x = 5;
```

- **uniform**: إذا كان متحول ما مسبوقاً بالكلمة المحجوزة uniform فهذا يعني أن المتحول سيهيأ خارج المظلل (مثلاً من قبل تطبيق مكتوب بلغة ++C). ويستخدم داخل المظلل.
- **extern**: إذا كان متحول ما مسبوقاً بالكلمة المحجوزة extern فهذا يعني أنه يمكن الولوج إلى المتحول من خارج المظلل (من قبل تطبيق مكتوب بلغة ++C مثلاً). المتحولات العامة فقط يمكن أن تكون مسبوقة بالكلمة المحجوزة extern. كما أن المتحولات العامة التي هي ليست static تكون extern افتراضياً.
- **shared**: إذا كان متحول ما مسبوقاً بالكلمة المحجوزة shared فهذا يشير لإطار عمل التأثيرات إلى أن المتحول سيكون مشتركاً بين عدة تأثيرات. المتحولات العامة فقط يمكن أن تكون مسبوقة بالكلمة المحجوزة shared.
- **volatile**: إذا كان متحول ما مسبوقاً بالكلمة المحجوزة volatile فهذا يشير لإطار عمل التأثيرات إلى أن هذا المتحول ستتغير قيمته كثيراً. المتحولات العامة فقط يمكن أن تكون مسبوقة بالكلمة المحجوزة volatile.
- **const**: إن للكلمة المحجوزة const في لغة HLSL نفس المعنى في لغة ++C. أي إذا كان متحول ما مسبوقاً بالكلمة المحجوزة const عندئذ سيبقى المتحول ثابتاً لا يمكن تغيير قيمته.

```
const float pi = 3.14f;
```

16.4: الكلمات المحجوزة والعبارات وقلب الأنواع

16.4.1: الكلمات المحجوزة

فيما يلي نورد الكلمات المحجوزة المعرفة في لغة HLSL:

asm	bool	compile	const	decl	do
double	else	extern	false	float	for
half	if	in	inline	inout	int

matrix	out	pass	pixelshader	return	sampler
shared	static	string	struct	technique	texture
true	typedef	uniform	vector	vertexshader	void
volatile	while				

فيما يلي نورد المعرفات المحجوزة ولكن غير المستخدمة والتي قد تصبح كلمات محجوزة مستقبلاً:

auto	break	case	catch	char	class
const_cast	continue	default	delete	dynamic_cast	enum
explicit	friend	goto	long	mutable	namespace
new	operator	private	protected	public	register
reinterpret_cast	short	signed	sizeof	static_cast	switch
template	this	throw	try	typename	union
unsigned	using	virtual			

16.4.2: انسياب البرنامج

توفر لغة HLSL العديد من عبارات لغة C++ المألوفة من أجل الاختيار والتكرار وانسياب البرنامج بشكل عام. إن صيغ هذه العبارات مشابهة تماماً لمثيلاتها في لغة C++:

- **عبارة Return:**

```
return (expression);
```

- **عبارتا If وIf...Else:**

```
if( condition )
{
    statement(s);
}

if( condition )
{
    statement(s);
}
else
{
    statement(s);
}
```

- **عبارة for:**

```
for(initial; condition; increment)
{
```

```
    statement(s);
}
```

- عبارة while:

```
while( condition )
{
    statement(s);
}
```

- عبارة do...while:

```
do
{
    statement(s);
} while ( condition );
```

16.4.3: قلب الأنواع

توفر لغة HLSL نموذجاً مرناً لقلب الأنواع. صيغة قلب الأنواع في لغة HLSL هي نفسها في لغة C. لقلب نوع float إلى matrix نكتب:

```
float f = 5.0f;
matrix m = (matrix)f;
```

يمكنك استخلاص معنى قلب النوع في أمثلة هذا الكتاب من خلال السياق. أما إذا أردت الحصول على معلومات مفصلة عن طرق قلب الأنواع المتوفرة فيمكنك أن تراها في وثائق مجموعة تطوير DirectX تحت البند Contents في:

```
DirectX Graphics\References\Shader Reference\High Level Shading
Language\Type.
```

16.5: العمليات

توفر لغة HLSL العديد من العمليات المألوفة في لغة C++. تستخدم هذه العمليات تماماً كما تستخدم في لغة C++ إلا في بعض الاستثناءات المذكورة أدناه. فيما يلي لائحة بالعمليات في HLSL:

```
[]     .      >      <      <=     >=
!=     ==     !      &&     ||     ?:
+      +=     -      -=     *      *=
/      /=     %      %=     ++     --
=      ()     ,
```

بالرغم من أن سلوك العمليات يشبه كثيراً سلوكها في لغة C++ إلا أنه توجد بعض

الفروق. أولاً تستخدم العملية % مع الأعداد الصحيحة وذات الفاصلة العائمة. كما يجب أن تكون القيمتان على يسار ويمين العملية من نفس الإشارة (أي يجب أن يكون كلا الطرفين موجبين أو سالبين).

ثانياً، انتبه إلى أن العديد من العمليات في HLSL تعمل على أساس كل مركبة على حدة. هذا ناتج عن كون العمليات تتم على مستوى المركبات لذا يمكن إجراء عمليات جمع الأشعة والمصفوفات وطرح الأشعة والمصفوفات واختبارات تساوي الأشعة والمصفوفات باستخدام نفس العمليات المستعملة مع الأنواع السلمية. انظر إلى الأمثلة التالية:

```
vector u = {1.0f, 0.0f, -3.0f, 1.0f};
vector v = {-4.0f, 2.0f, 1.0f, 0.0f};

vector sum = u + v; // sum = (-3.0f, 2.0f, -2.0f, 1.0f)
```

زيادة شعاع من خلال العملية ++ تزيد قيمة كل مركبة:

```
// before increment: sum = (-3.0f, 2.0f, -2.0f, 1.0f)
sum++; // after increment: sum = (-2.0f, 3.0f, -1.0f, 2.0f)
```

ضرب المركبات:

```
vector u = {1.0f, 0.0f, -3.0f, 1.0f};
vector v = {-4.0f, 2.0f, 1.0f, 0.0f};

vector product = u * v; // product = (-4.0f, 0.0f, -3.0f, 0.0f)
```

تتم المقارنة بتطبيق عمليات المقارنة على كل مركبة على حدة، وتعيد شعاعاً (أو مصفوفة) كل مركبة فيه من النوع bool. حيث يحتوي الشعاع البولياني الناتج على نتائج مقارنة كل مركبة على حدة. مثلاً:

```
vector u = {1.0f, 0.0f, -3.0f, 1.0f};
vector v = {-4.0f, 0.0f, 1.0f, 1.0f};

vector b = (u == v); // b = (false, true, false, true)
```

أخيراً، سنختتم حديثنا بدراسة ترقية المتحول في العمليات الثنائية (ذات الحدين):

- بالنسبة إلى العمليات الثنائية إذا كان الطرف الأيسر يختلف عن الطرف الأيمن في الأبعاد عندئذ تتم ترقية الطرف ذو الأبعاد الأصغر (يتم قلب نوعه) ليصير بنفس

أبعاد الطرف ذو الأبعاد الأكبر. مثلاً: إذا كان x من النوع float وy من النوع float3 عندها تتم ترقية x في التعبير (x+y) إلى النوع float3 والتعبير سيعطي قيمة من النوع float3. يتم إنجاز الترقية عن طريق قلب النوع المعرف. ففي هذه الحالة نقوم بقلب نوع سلمي إلى شعاع، لذلك بعد أن تتم ترقية x إلى float3 يكون x = (x, x, x) حسب تعريف قلب النوع السلمي إلى شعاع. لاحظ أن الترقية تكون غير معرفة إذا لم يكن قلب النوع معرفاً. مثلاً، لا يمكن ترقية float2 إلى float3 لأنه لا يوجد قلب أنواع معرف كهذا.

- بالنسبة إلى العمليات الثنائية، إذا كان الطرف الأيسر يختلف عن الطرف الأيمن في النوع، عندئذ تتم ترقية (قلب نوع) الطرف ذو النوع الأقل دقة ليصير نفس نوع الطرف ذو النوع الأكبر دقة. مثلاً، إذا كان x من النوع int وy من النوع half، فتتم ترقية المتحول x ضمن التعبير (x+y) إلى النوع half وتكون قيمة التعبير الناتجة من النوع half.

16.6: التوابع المعرفة من قبل المستخدم

إن للتوابع في لغة HLSL الخصائص التالية:

- تستخدم التوابع صيغة ++C المألوفة.
- يتم تمرير الوسطاء عن طريق القيمة (by value) فقط.
- التعاودية غير مسموحة.
- تستخدم التوابع دائماً بشكل inline.

وفوق كل ذلك تضيف لغة HLSL بعض الكلمات المحجوزة الإضافية التي تستخدم مع التوابع. على سبيل المثال، تأمل التابع التالي المكتوب بلغة HLSL:

```
bool foo(in const bool b, // input bool
         out int r1,      // output int
         inout float r2)  // input/output float
{
    if( b )               // test input value
    {
        r1 = 5;           // output a value through r1
    }
    else
    {
```

```
        r1 = 1;            // output another value through r1
    }

    // since r2 is inout we can use it as an input value
    // and also output a value through it
    r2 = r2 * r2 * r2;

    return true;
}
```

إن هذا التابع مماثل تقريباً لأي تابع بلغة C++ فيما عدا وجود الكلمات المحجوزة in وout وinout.

- **in**: يعني أنه ستنسخ قيمة الوسيط الفعلي إلى الوسيط الشكلي قبل بدء تنفيذ التابع. ليس من الضروري ذكر in قبل اسم الوسيط صراحة لأن الوسيط هو in افتراضياً. مثلاً، إن كلا التابعين التاليين متكافئين:

```
float square(in float x)
{
    return x * x;
}
```

(بدون ذكر in بشكل صريح):

```
float square(float x)
{
    return x * x;
}
```

- **out**: يعني أنه سيُنسخ الوسيط الشكلي إلى الوسيط الفعلي عندما يعود التابع. يفيدنا هذا الأمر من أجل إعادة القيم من خلال الوسطاء. إن الكلمة المحجوزة out ضرورية لأن لغة HLSL لا تسمح بتمرير الوسطاء مرجعياً كما لا تسمح بتمرير المؤشرات. نلاحظ أنه لما يكون الوسيط المحدد على أنه out، لا يُنسخ الوسيط الفعلي إلى الوسيط الشكلي قبل بدء التابع، وبالتالي لا يمكننا افتراض وجود قيمة أولية فيه ذات معنى قبل أن نقوم بإسناد قيمة ما إليه. بكلام آخر: يستخدم وسيط ذو out فقط من أجل إخراج المعطيات ولا يستخدم أبداً كدخل.

```
void square(in float x, out float y)
{
    y = x * x;
}
```

هنا ندخل العدد الذي نريد تربيعه من خلال الوسيط x ونأخذ القيمة المعادة (مربع x) من خلال الوسيط y.

- **inout**: يعني بأن الوسيط هو in وout في نفس الوقت. استعمل inout إذا أردت أن تستخدم وسيطاً ما كدخل وخرج معاً.

```
void square(inout float x)
{
    x = x * x;
}
```

هنا ندخل العدد الذي نريد تربيعه من خلال الوسيط x، ونحصل على النتيجة أيضاً من خلال الوسيط x.

16.7: التوابع الضمنية

إن لغة HLSL غنية بالتوابع الضمنية المفيدة في الرسوميات ثلاثية الأبعاد. الجدول التالي عبارة عن لائحة موجزة من هذه التوابع. سنتمرن على استخدام بعض هذه التوابع في الفصلين التاليين. أما حالياً فقم بالتعرف إليها فقط.

ملاحظة: يمكن الحصول على لائحة كاملة بتوابع HLSL الضمنية من وثائق DirectX تحت البند Contents ثم

```
DirectX Graphics\Reference\Shader Reference\High Level
Shader Language\Intrinsic Functions.
```

التابع	وصفه
abs(x)	يعيد |x|.
ceil(x)	يعيد أصغر عدد صحيح أكبر أو يساوي x.
clamp(x, a, b)	يجعل x ضمن المجال [a, b] إذا كان خارجه (إذا كان x > b يصير x = b وإذا كان x < a يصير x = a) ويعيد الناتج.
cos(x)	يعيد جيب التمام للزاوية x (حيث x مقدرة بالراديان).
cross(u, v)	يعيد u×v.
degrees(x)	يحول x من الراديان إلى الدرجات.
determinant(M)	يعيد قيمة محدد المصفوفة M (أي (det(M)).
distance(u, v)	يعيد المسافة ||u-v|| بين النقطتين v, u.
dot(u, v)	يعيد u.v.
floor(x)	يعيد أكبر عدد صحيح أصغر أو يساوي x.

التابع	وصفه
length(v)	يعيد ||v||.
lerp(u, v, t)	يوجد التركيب الخطي لكل من u وv اعتماداً على الوسيط t حيث $t\in [0,1]$
log(x)	يعيد ln(x).
log10(x)	يعيد $\log_{10}(x)$.
log2(x)	يعيد $\log_{2}(x)$.
max(x, y)	يعيد القيمة العظمى من القيمتين x ,y.
min(x, y)	يعيد القيمة الصغرى من القيمتين x ,y.
mul(M, N)	يعيد جداء المصفوفتين MN. لاحظ أنه يجب أن يكون الجداء MN معرفاً. إذا كان M عبارة عن شعاع عندئذ يُعامل كشعاع سطر بشرط أن يكون جداء الشعاع بمصفوفة معرفاً. كذلك إذا كان N عبارة عن شعاع عندئذ يُعامل كشعاع عمود بشرط أن يكون جداء المصفوفة بشعاع معرفاً.
normalize(v)	يعيد v/||v||.
pow(b, n)	يعيد b^n.
radians(x)	يحول x من الدرجات إلى الراديان.
reflect(v, n)	يوجد شعاع الانعكاس اعتباراً من شعاع الورود v وناظم السطح n.
refract(v, n, eta)	يوجد شعاع الانكسار اعتباراً من شعاع الورود v وناظم السطح n وعامل الانكسارالنسبي بين المادتين eta.
rsqrt(x)	يعيد $1/\sqrt{x}$
saturate(x)	يعيد clamp(x, 0.0, 1.0).
sin(x)	يعيد جيب الزاوية x (حيث x مقدرة بالراديان).
sincos(in x, out s, out c)	يعيد جيب وجيب التمام للزاوية x (حيث x مقدرة بالراديان).
sqrt(x)	يعيد $\sqrt{x}$.
tan(x)	يعيد ظل الزاوية x (حيث x مقدرة بالراديان).
transpose(M)	يعيد المنقول M^T.

قد أُجري التحميل الزائد على معظم هذه التوابع بغية استخدامها مع كل الأنواع الضمنية الممكنة. مثلاً يستخدم التابع abs مع كل الأنواع السلمية، أما الجداء الشعاعي cross فيستخدم فقط مع الأشعة ثلاثية الأبعاد من أي نوع (أي أشعة ثلاثية الأبعاد مركباتها int أو float... الخ).

أما بالنسبة إلى التابع lerp فإنه يُستخدم مع القيم السلمية والأشعة ثنائية وثلاثية ورباعية

الأبعاد وبالتالي يتم إجراء التحميل الزائد لها من أجل كل الأنواع.

إذا قمت بتمرير قيمة غير سلمية إلى تابع ذو وسيط سلمي (أي: تابع يأخذ عادة وسيطاً سلمياً مثل cos(x)) عندها سيعمل التابع على كل مركبة على حدة. مثلاً إذا كتبت:

```
float3 v = float3(0.0f, 0.0f, 0.0f);

v = cos(v);
```

عندئذ سيكون: v = (cos(x), cos(y), cos(z)).

توضح الأمثلة التالية كيف يتم استدعاء بعضاً من التوابع السابقة:

```
float x = sin(1.0f);
float y = sqrt(4.0f);

vector u = {1.0f, 2.0f, -3.0f, 0.0f};
vector v = {3.0f, -1.0f, 0.0f, 2.0f};
float s = dot(u, v);

float3 i = {1.0f, 0.0f, 0.0f};
float3 j = {0.0f, 1.0f, 0.0f};
float3 k = cross(i, j);

matrix<float, 2, 2> M = {1.0f, 2.0f, 3.0f, 4.0f};
matrix<float, 2, 2> T = transpose(M);
```

16.8: الخلاصة

- تُكتب برامج لغة HLSL في ملفات نصية ASCII ويتم ترجمتها في تطبيقاتنا باستخدام التابع D3DXCompileShaderFromFile.
- تسمح لنا الواجهة ID3DXConstantTable بتحديد قيم متحولات برنامج المظلل من داخل التطبيق. عملية الاتصال ضرورية لأن المتحولات في المظلل قد تتغير من إطار إلى إطار. مثلاً، إذا تغيرت مصفوفة العرض في التطبيق، عندها سنحتاج إلى تحديث متحول مصفوفة العرض في المظلل. حيث يمكننا فعل ذلك من خلال ID3DXConstantTable.
- يجب تعريف بنيتا دخل وخرج من أجل كل مظلل نكتبه وذلك لوصف تنسيق المعطيات التي تدخل إلى المظلل والتي تخرج منه.

- يوجد لكل مظلل تابع نقطة دخول يأخذ وسيطاً عبارة عن بنية دخل تُستخدم لتمرير معطيات الدخل إلى المظلل. كما يعيد كل مظلل بنية رأس تُستخدم لإخراج المعطيات من المظلل.